

**MYMPI**

**MYMPI**

Copyright © 2008 San Diego Super Computer Center, University of California - San Diego

# Table of Contents

<b>Preface</b> .....	<b>i</b>
<b>1. Welcome</b> .....	<b>1</b>
1.1. Introduction .....	1
1.2. Authors .....	1
<b>2. MYMPI Questions</b> .....	<b>2</b>
2.1. What is MyMPI? .....	2
2.2. Are there other MPIs for Python? .....	2
2.3. How is it different? .....	2
2.4. What routines are available? .....	2
2.5. What data types? .....	4
2.6. What arguments are needed for the routines? .....	4
2.7. So how do we mix parallel Python with Fortran and C? .....	4
2.8. Can this be used as a teaching tool? .....	5
2.9. Who supported this work? .....	5
2.10. Can I see it in action? .....	5
<b>3. Getting MYMPI</b> .....	<b>7</b>
3.1. Download Package .....	7
3.2. Getting Source Code .....	7
<b>4. Install MYMPI</b> .....	<b>8</b>
4.1. Required Software .....	8
4.2. Installation Procedure .....	8
<b>5. Programming with MYMPI</b> .....	<b>9</b>
5.1. MYMPI API .....	9
5.2. Running MYMPI Programs .....	9
5.3. MYMPI Example 1 .....	9
5.4. MYMPI Example 2 .....	10
5.5. MYMPI Example 3 .....	11
<b>6. Recent Changes</b> .....	<b>13</b>
6.1. Version 1.15 Changes .....	13
6.2. Version 1.14 Changes .....	13
6.3. Version 1.13 Changes .....	13
6.4. Version 1.1 Release Notes .....	14
6.5. Other Recent Changes .....	14
<b>7. Version 1.13 Release Notes</b> .....	<b>15</b>
7.1. Recent Bug Fixes .....	15
7.2. New MPI-1 routines .....	15
7.3. Attributes for use with <code>mpi_attn_get</code> .....	15
7.4. MPI Version .....	16
7.5. MPI-2 routines for launching new MPI processes .....	16
7.6. Arguments for <code>mpi_comm_spawn</code> .....	16
7.7. A simple MPI-2 example using <code>mpi_comm_spawn</code> .....	17
7.8. Example Output .....	22
7.9. A simple MPI-2 example using two independent programs .....	22

<b>8. Version 1.11 Release Notes</b> .....	<b>25</b>
8.1. Introduction .....	25
8.2. Long command line argument lists are allowed.....	25
8.3. <b>mpi_init</b> now returns the command line arguments .....	25
8.4. Contains a version number .....	25
8.5. Contains a work flow example .....	25

# List of Examples

5-1. example1.py.....	9
5-2. example2.py.....	10
5-3. example1.py.....	11
7-1. tspawn.py.....	18
7-2. worker.py.....	19
7-3. worker.c.....	20
7-4. driver program for tspawn.....	21
7-5. main0.py.....	23
7-6. main1.py.....	23

# Preface

Welcome to MYMPI!

- Want an easy way to mix Fortran and/or C with Python?
- Want to use your standard Python interpreter to:
  - write parallel Python programs?
  - write front ends for parallel Fortran or C programs?
  - take advantage of multiple processors on your desktop machine?
- Want to teach or learn parallel programming using Python?

# Chapter 1. Welcome

## 1.1. Introduction

MYMPI is a Python module that works with standard Python interpreter. It implements an important subset of the standard parallel programming library, MPI or Message Passing Interface<sup>1</sup>.

## 1.2. Authors

- Timothy H. Kaiser, PhD <tkaiser@sdsc.edu>.

## Notes

1. <http://www-unix.mcs.anl.gov/mpi/>

# Chapter 2. MYMPI Questions

## 2.1. What is MyMPI?

MYMPI is a Python module that allows parallel programming using the Message Passing Interface or MPI<sup>1</sup>. It allows creating pure Python parallel programs as well as mixing Python programs with Fortran or C programs.

## 2.2. Are there other MPIs for Python?

Yes. The best know in pyMPI at <http://sourceforge.net/projects/pympi/>.

## 2.3. How is it different?

The main difference between pyMPI and MYMPI is that pyMPI is actually a special version of the Python interpreter along with a module. Our version, MYMPI, is a module only. MYMPI is used with a normal Python interpreter.

Those familiar with "normal" MPI know that the routine **MPI\_Init** is used to initialize a MPI program. Every process in the parallel job calls **MPI\_Init**. pyMPI departs from this standard operating procedure. pyMPI programs do not call **MPI\_Init**. **MPI\_Init** is built into the interpreter. That is, **MPI\_Init** is called when the interpreter is launched. Calling **MPI\_Init** is implicit. In MYMPI, programs explicitly call **MPI\_Init**.

There is a fundamental difference in the semantics of the two packages. With pyMPI the interpreter is the parallel application. With MYMPI the code executed by the interpreter is the parallel application. One of the reasons why MYMPI was created was to provide better control of how and when **MPI\_Init** is called.

MYMPI is much smaller since it is only a module not a full interpreter. It builds faster and in theory is easier to port to a new platform. It is also smaller because it only implements about 30 of the 120+ MPI calls.

The syntax of the calls in our package match the syntax of C and Fortran calls more closely than the syntax of pyMPI calls match C and Fortran. For example, with MYMPI arguments are explicit. With pyMPI arguments can be implicit in some instances. pyMPI has a more OOP flavor. We chose to match C and Fortran more closely because we intended from the start that we would be writing programs that mixed Python with the other languages.

## 2.4. What routines are available?

The following routines are available in version 1.11:

- **mpi\_alltoall** - Sends data from all to all processes
- **mpi\_alltoallv** - Sends data from all to all processes, with a displacement
- **mpi\_barrier** - Blocks until all process have reached this routine.
- **mpi\_bcast** - Broadcasts a message from the process with rank "root" to all other processes of the group.
- **mpi\_comm\_create** - Creates a new communicator

- **mpi\_comm\_dup** - Duplicates an existing communicator with all its cached information
- **mpi\_comm\_group** - Accesses the group associated with given communicator
- **mpi\_comm\_rank** - Determines the rank of the calling process in the communicator
- **mpi\_comm\_size** - Determines the size of the group associated with a communicator
- **mpi\_comm\_split** - Creates new communicators based on colors and keys
- **mpi\_error** - checks for errors (not part of regular MPI)
- **mpi\_finalize** - Terminates MPI execution environment
- **mpi\_gather** - Gathers together values from a group of processes
- **mpi\_gatherv** - Gathers into specified locations from all processes in a group
- **mpi\_get\_count** - Gets the number of "top level" elements
- **mpi\_group\_incl** - Produces a group by reordering an existing group and taking only listed members
- **mpi\_group\_rank** - Returns the rank of this process in the given group
- **mpi\_init** - Initialize the MPI execution environment
- **mpi\_iprobe** - Nonblocking test for a message
- **mpi\_probe** - Blocking test for a message
- **mpi\_recv** - Basic receive
- **mpi\_reduce** - Reduces values on all processes to a single value
- **mpi\_scatter** - Sends data from one task to all other tasks in a group
- **mpi\_scatterv** - Scatters a buffer in parts to all tasks in a group
- **mpi\_send** - Performs a basic send
- **mpi\_status** - Return status information (not part of standard MPI)

New MPI-1 routines were added in version 1.12.

- **mpi\_wtime** - Returns a time (float) in seconds since an arbitrary time in the past.
- **mpi\_wtick** - Return value Time in seconds of resolution of mpi\_wtime
- **mpi\_get\_processor\_name** - Returns a (string) unique specifier (name) for the node on which it is called
- **mpi\_attr\_get** - Returns a value associated with some attribute.

The following MPI-2 dynamic process creation related routines are available in version 1.13.

- **mpi\_comm\_spawn** - Spawns additional MPI tasks
- **mpi\_comm\_get\_parent** - Returns the communicator associated with the parent for a newly spawned task
- **mpi\_comm\_free** - Returns a communicator associated with spawned tasks.
- **mpi\_intercomm\_merge** - Returns an intracommunicator from an intercommunicator
- **mpi\_open\_port** - Establish a port at which an application may be contacted
- **mpi\_close\_port** - Releases the network address represented by port\_name
- **mpi\_comm\_accept** - Accept connections from clients

- **mpi\_comm\_connect** - Establishes communication with a server specified by port\_name
- **mpi\_comm\_disconnect** - Terminates communication with a server specified by a communicator

## 2.5. What data types?

- **MPI\_INT** - Integer
- **MPI\_DOUBLE** - 8 byte floating point number

We have support for scalars and multiple dimension arrays using the Numeric package. The package can also be built using NumPy instead of Numeric. See the README file.

## 2.6. What arguments are needed for the routines?

The input argument lists closely match the list for C and Fortran MPI routines. The difference between the Python call and C/Fortran calls is that values returned via assignment. For example, in C we would do a message receive into *buffer* of *count* integers from process *source* using the syntax:

```
MPI_Recv(&buffer, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
```

In Python this would be:

```
buffer=mpi.mpi_recv(count, mpi.MPI_INT, source, tag, mpi.MPI_COMM_WORLD)
```

After these calls *buffer* would contain the data from the remote process. Status is returned using a special call

```
status=mpi_status()
```

After this call status is an array that contains, the source of the message, the tag, and the error code.

## 2.7. So how do we mix parallel Python with Fortran and C?

Simply stated, you write a MPI program in Python, one in Fortran and/or C and run them on your parallel machine. The problem is in the details. Most MPI systems have a command such as, mpirun, that launches mpi jobs. The exact method of launching heterogeneous MPI jobs is machine specific. We will give some examples. We will assume a python application `hello_python` and a fortran application `hello_f`.

For the *IBM SP running poe* the script setup is simple. First, set the environmental variable **MP\_PGMMODEL** to **MPMD**. You write your script as you normally would but you don't give the name of the executable on the poe command line. You list the names of the executables after the poe command as shown below.

```
setenv MP_PGMMODEL MPMD
```

```
poe << HERE
```

```

/users01/sdsc/tkaiser/hello_python
/users01/sdsc/tkaiser/hello_f
HERE

```

For machines using MPICH and/or pbs the procedure is a little more complicated. The method is outlined here<sup>3</sup>.

## 2.8. Can this be used as a teaching tool?

Yes, not only for Python and MPI but MPI in Fortran and C. MYMPI calls were designed to match calls in Fortran in C. On the other hand, pyMPI calls have a much more object oriented flavor. While this is fine for pure Python programmers MYMPI is easier to blend with Fortran and C. In fact, a direct translation of a pyMPI program to C or Fortran would not run because pyMPI programs do not call **MPI\_Init**;

The examples that are part of the distribution are actually rewrites of C and Fortran examples that are used in teaching introductory MPI. There is a series of slides that can be used with the examples.

## 2.9. Who supported this work?

This work was done as part of the National Biomedical Computation resource project, NBCR<sup>4</sup>. NBCR is supported by the National Institutes of Health (NIH) through a National Center for Research Resources program grant (P 41 RR08605) to researchers at the University of California, San Diego, including the San Diego Supercomputer Center (SDSC), the California Institute of Telecommunications and Information Technology (Cal-(IT)2), the Center for Research on Biological Structure (CRBS), The Scripps Research Institute (TSRI), and Washington University in St. Louis.

## 2.10. Can I see it in action?

Our example will show a MPI application that has three processes or tasks. The processes are a mixture of Python, Fortran and C. The source for this example is distributed along with the source for the Python module.



5

## Notes

1. <http://www-unix.mcs.anl.gov/mpi/>

2. <http://sourceforge.net/projects/pympi/>
3. <http://peloton.sdsc.edu/~tkaiser/mympi/heter/index.html>
4. <http://nbcv.sdsc.edu/>
5. `movies/Terminal.mov`

# Chapter 3. Getting MYMPI

## 3.1. Download Package

Visit our sourceforge page<sup>1</sup> to download binaries.

## 3.2. Getting Source Code

Use the following command to check out a copy of our code from the SVN repository.

```
svn co https://pydusa.svn.sourceforge.net/svnroot/pydusa pydusa
```

Visit our sourceforge page for SVN access<sup>2</sup> for more information.

## Notes

1. [http://sourceforge.net/project/showfiles.php?group\\_id=218633](http://sourceforge.net/project/showfiles.php?group_id=218633)
2. [http://sourceforge.net/svn/?group\\_id=218633](http://sourceforge.net/svn/?group_id=218633)

# Chapter 4. Install MYMPI

## 4.1. Required Software

1. MPI C Compiler
2. Python (MYMPI has been tested using versions 2.3 and 2.4)
3. Numeric (used to compile certain example programs)
4. Numpy (used to compile certain example programs)
5. MPI Fortran 90 Compiler (OPTIONAL: used to compile example Fortran programs)

## 4.2. Installation Procedure

1. **./configure**
2. **make**
3. **make install**
4. **make test (optional)**

Some of the useful **configure** options are:

- **--with-python** - location of the python executable
- **--with-mpicc** - location of mpicc
- **--with-mpif90** - location of mpif90

# Chapter 5. Programming with MYMPI

## 5.1. MYMPI API

Click [here](#)<sup>1</sup> to see MYMPI API documentation.

## 5.2. Running MYMPI Programs

You should run an MYMPI program the same way you run MPI programs.

```
mpirun -np $num_nodes $executable_name
```

For example,

```
mpirun -np 5 p_ex02.py
```

Output

```
hello from 0 of 5
arg_str ('/Users/jren/mympi/mpi_tests/p_ex02.py',)
hello from 1 of 5
hello from 2 of 5
hello from 4 of 5
hello from 3 of 5
getting 2
i= [1234 5678]
```

## 5.3. MYMPI Example 1

In this program, each node prints out it's rank and the size of the current MPI run (total number of nodes). In addition, each node prints out the same of the python program with a message "python is not about snakes".

### Example 5-1. example1.py

```
#!/usr/bin/env python
import Numeric
from Numeric import *
import mpi
from mpi import *
import sys

#print "before",len(sys.argv),sys.argv
sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
print "after ",len(sys.argv),sys.argv

myid=mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
```

```

numprocs=mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)

print "Hello from ",myid
print "Numprocs is ",numprocs

print "python is not about snakes"

###print sys.executable,sys.path

mpi.mpi_finalize()

```

Sample output using 2 nodes:

```

after 1 ('/Users/jren/mympi/mpi_tests/p_ex00.py',)
Hello from 0
Numprocs is 2
python is not about snakes
after 1 ('/Users/jren/mympi/mpi_tests/p_ex00.py',)
Hello from 1
Numprocs is 2
python is not about snakes

```

## 5.4. MYMPI Example 2

This is a simple send/receive program in MPI. Each node prints out `hello from` its rank and the size of the current MPI run (total number of nodes). Then, node 0 sends a message to node 1. Finally node 1 receives the message.

### Example 5-2. `example2.py`

```

#!/usr/bin/env python
import Numeric
from Numeric import *
import mpi
import sys

#print "before",len(sys.argv),sys.argv
sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
#print "after ",len(sys.argv),sys.argv

myid=mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
numprocs=mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)
print "hello from ",myid," of ",numprocs

tag=1234
source=0
destination=1
count=1
if myid == source:
    buffer=5678

```

```

buffer=array([5678]),"i")
mpi.mpi_send(buffer, count, mpi.MPI_INT,destination,tag, mpi.MPI_COMM_WORLD)
print "processor ",myid," sent ",buffer

if myid == destination:
    buffer=mpi.mpi_recv(count, mpi.MPI_INT,source,tag, mpi.MPI_COMM_WORLD)
    print "processor ",myid," got ",buffer

mpi.mpi_finalize()

```

Sample output using 2 nodes:

```

hello from 0 of 2
hello from 1 of 2
processor 0 sent [5678]
processor 1 got [5678]

```

## 5.5. MYMPI Example 3

This program shows how to use MPI\_Scatter and MPI\_Reduce Each processor gets different data from the root processor by way of mpi\_scatter. The data is summed and then sent back to the root processor using MPI\_Reduce. The root processor then prints the global sum.

### Example 5-3. example1.py

```

#!/usr/bin/env python

import Numeric
from Numeric import *
import mpi
import sys

#print "before",len(sys.argv),sys.argv
sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
#print "after ",len(sys.argv),sys.argv

myid=mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
numnodes=mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)
print "hello from ",myid," of ",numnodes

mpi_root=0

#each processor will get count elements from the root
count=4
# in python we do not need to preallocate the array myray
# we do need to assign a dummy value to the send_ray
send_ray=zeros(0,"i")

```

```

if myid == mpi_root:
    size=count*numnodes;
    send_ray=zeros(size,"i")
    for i in range(0, size):
        send_ray[i]=i

#send different data to each processor
myray=mpi.mpi_scatter(send_ray,count,mpi.MPI_INT,count,mpi.MPI_INT,mpi_root,mpi.MPI_COMM_WORLD)

#each processor does a local sum
total=0
for i in range(0, count):
    total=total+myray[i]
print "myid=",myid,"total=",total

#reduce back to the root and print
back_ray=mpi.mpi_reduce(total,1, mpi.MPI_INT,mpi.MPI_SUM,mpi_root,mpi.MPI_COMM_WORLD)
if myid == mpi_root:
    print "results from all processors=",back_ray

mpi.mpi_finalize()

```

Sample output using 3 processors

```

hello from 1 of 3
hello from 0 of 3
hello from 2 of 3
myid= 0 total= 6
myid= 1 total= 22
myid= 2 total= 38
results from all processors= [66]

```

## Notes

1. mpi.html

# Chapter 6. Recent Changes

## 6.1. Version 1.15 Changes

- Added autoconf/automake
- Added **make install**
- Added **make test**
- Added Improved documentation
- Automatically configures various paths
- `configure` automatically what can be compiled based on installed software
- Restructured the package

## 6.2. Version 1.14 Changes

`makefile.leopard`

- OSX 10.5/Intel makefile

Compile time option

- `-DNULL_INIT` will suppress passing arguments to `MPI_Init`, needed for some versions of mpi (MPICH2 on OSX for example)

Source code additions

- Some debugging code and hooks for future development.

## 6.3. Version 1.13 Changes

Version 1.13 has support for the following MPI2 routines:

- `mpi_comm_spawn` - Spawns additional MPI tasks
- `mpi_comm_get_parent` - Returns the communicator associated with the parent for a newly spawned task
- `mpi_comm_free` - Returns a communicator associated with spawned tasks.
- `mpi_intercomm_merge` - Returns an intracommunicator from an intercommunicator
- `mpi_open_port` - Establish a port at which an application may be contacted
- `mpi_close_port` - Releases the network address represented by `port_name`

- **mpi\_comm\_accept** - Accept connections from clients
- **mpi\_comm\_connect** - Establishes communication with a server specified by port\_name
- **mpi\_comm\_disconnect** - Terminates communication with a server specified by a communicator
- **mpi\_comm\_set\_errhandler** - Sets the behavior if an MPI error occurs using the given communicator

## 6.4. Version 1.1 Release Notes

You can find version 1.1 release notes here<sup>1</sup>.

## 6.5. Other Recent Changes

1. Bug fix for gatherv (Jan 12, 07)
2. Bug fix for scatter (Dec 8, 06)
3. Added the routine **mpi\_comm\_set\_errhandler** (Nov 29)
4. Bug fix for **mpi\_send**, for multidimensional arrays (Nov 29)
5. Minor changes to doc string include file for AIX compiler.
6. Version 1.13 collectives now work for dynamically created processes.
7. Version 1.13 added doc strings (on line documentation) for all routines.
8. Version 1.12.2 fixes a potentially serious memory leak in **mpi\_send** and less serious in **mpi\_group\_incl**.
9. Version 1.12.1 fixes problems with non-integers in **mpi\_scatter**, **mpi\_scatterv**, **mpi\_alltoall**.
10. Added notes on building using NumPy instead of Numeric.
11. Added makefile for IBM "SP class" machines.

## Notes

1. [http://peloton.sdsc.edu/~tkaiser/mympi/mympimod\\_1.11/release1.11\\_notes.html](http://peloton.sdsc.edu/~tkaiser/mympi/mympimod_1.11/release1.11_notes.html)

# Chapter 7. Version 1.13 Release Notes

## 7.1. Recent Bug Fixes

- Memory leak in **mpi\_send** (ouch) and **mpi\_group\_incl**
- Multidimensional arrays were not working for **mpi\_send**
- Collectives were not working correctly for MPI-2 dynamically created processes.

Version 1.12 added support for a number of additional routines and constants. The most significant additions are related to the support of dynamic MPI process creation. Version 1.13 adds even more support dynamic MPI process creation. These routines are available on systems that support MPI-2. The module can now return the MPI version number so the user can determine if the MPI-2 routines are supported.

Also, some additional MPI-1 routines were added in 1.12. These routines will be discussed first.

## 7.2. New MPI-1 routines

This additional MPI-1 routines are:

- **mpi\_wtime** - Returns a time (float) in seconds since an arbitrary time in the past.
- **mpi\_wtick** - Return value Time in seconds of resolution of **mpi\_wtime**
- **mpi\_get\_processor\_name** - Returns a (string) unique specifier (name) for the node on which it is called
- **mpi\_attr\_get** - Returns a value associated with some attribute.

## 7.3. Attributes for use with **mpi\_attr\_get**

There are several predefined attributes in MPI. Their "keys" have been added to the module as constants. These include:

- **MPI\_TAG\_UB** - Largest tag value
- **MPI\_HOST** - Rank of process that is host, if any
- **MPI\_IO** - Rank of process that can do I/O
- **MPI\_WTIME\_IS\_GLOBAL** - Has value 1 if **MPI\_WTIME** is globally synchronized.

The following shows how these attributes associated with these keys can be found:

```
#test attributes
print "MPI_WTIME_IS_GLOBAL",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_WTIME_IS_GLOBAL)

print "          MPI_TAG_UB",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_TAG_UB)
```

```
print "          MPI_HOST",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_HOST)
print "          MPI_IO",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_IO)
```

The Python implementation of **mpi\_attr\_get** is slightly different than the C or Fortran implementation. The "normal" versions also return a flag that indicates if the attribute about which is being inquired is defined. For the Python version, if you pass an undefined key to **mpi\_attr\_get** it will return NULL. (There is a bug in some versions of MPI that will cause this routine to crash if you pass in an undefined key.)

## 7.4. MPI Version

To print the **MPI\_VERSION** and **MPI\_SUBVERSION** you can do the following:

```
print "%s%1.1d%s%1.1d" % ("mpi version ",mpi.MPI_VERSION,".",mpi.MPI_SUBVERSION)
```

## 7.5. MPI-2 routines for launching new MPI processes

A number of MPI-2 routines were added to the module to support dynamic process creation. The makefile will detect (hopefully) if the library that you are using to build the module contains the MPI-2 routines. Note that not all parallel systems support dynamic process creation even if the library has the functionality. These routines have been tested with openmpi version 1.03(beta) and MPICH2 version 1.0.3 under using the default job launcher that ships with the MPI libraries.

The following MPI-2 related routines are now supported:

- **mpi\_comm\_spawn** - Spawns additional MPI tasks
- **mpi\_comm\_get\_parent** - Returns the communicator associated with the parent for a newly spawned task
- **mpi\_comm\_free** - Returns a communicator associated with spawned tasks.
- **mpi\_intercomm\_merge** - Returns an intracommunicator from an intercommunicator
- **mpi\_open\_port** - Establish a port at which an application may be contacted
- **mpi\_close\_port** - Releases the network address represented by port\_name
- **mpi\_comm\_accept** - Accept connections from clients
- **mpi\_comm\_connect** - Establishes communication with a server specified by port\_name
- **mpi\_comm\_disconnect** - Terminates communication with a server specified by a communicator
- **mpi\_comm\_set\_errhandler** - Sets the behavior if an MPI error occurs using the given communicator

There is also support for the attribute **MPI\_UNIVERSE\_SIZE**. This attribute is the number of processors that are available for spawning new tasks. This can be printed using the following:

```
print "  MPI_UNIVERSE_SIZE",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_UNIVERSE_SIZE)
```

## 7.6. Arguments for `mpi_comm_spawn`

### **command**

- name of program to be spawned (string, significant only at root)

### **argv**

- arguments to command (array of strings, significant only at root)

### **maxprocs**

- maximum number of processes to start (integer, significant only at root)

### **info**

- a set of key-value pairs telling the runtime system where and how to start the processes. This is not yet fully supported.

### **rank**

- rank of process in which previous arguments are examined

### **comm**

- intracommunicator containing group of spawning processes
- The output from `mpi_comm_spawn` is an intercommunicator between original group and the `mpi_comm_spawn` returns an intercommunicator between original group and the newly spawned group.
- There is an auxiliary routine, `mpi_array_of_errcodes`, that is not part of MPI.

### **mpi\_array\_of\_errcodes()**

- Returns an array of error codes associated with launching the new tasks.
- There are some constants that can be used with `mpi_comm_spawn`:

#### **MPI\_ARGV\_NULL**

- Passing an argv of `MPI_ARGV_NULL` to `MPI_COMM_SPAWN` results in main receiving argc of 1 and an argv whose element 0 is (conventionally) the name of the program.

#### **MPI\_INFO\_NULL**

- Used in place of the info key-value pairs.
- MPI-2 defines a constant `MPI_ROOT` that can be used with `MPI_Comm_spawn`. I have found using `MPI_ROOT` can cause problems. In some instances `MPI_Comm_spawn` will return an error. Use `MPI_ROOT` with caution. This has been reported to the developers.
- The Python version of `mpi_comm_spawn` will also take a zero length string in place of argv. This is equivalent to passing `MPI_ARGV_NULL`. Also, if you pass in a single string, this is equivalent to passing in an array of strings. The single string gets replicated and passed to each new process.

## 7.7. A simple MPI-2 example using `mpi_comm_spawn`

The program, `tspawn.py`, shows how the new routines can be used. It prints a number of constants and attributes. It then uses `mpi_comm_spawn` to launch a 3 instances of second Python MPI program, `worker.py`. Each instance of this second program prints creates a file with that contains a date stamp.

The program `tspawn.py` and `worker.py` engage in communications, Broadcast, Scatter, Send, Receive, and Reduction.

Finally, the `tspawn.py` program also starts a "C" version of the worker program. The source for these programs can be found in `mpi_tests/mpi2Examples`.

### Example 7-1. `tspawn.py`

```
#!/usr/bin/env python
import numpy
from numpy import *
import mpi
import sys
from time import sleep
from os import getcwd

sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)

t1=mpi.mpi_wtime()
tick=mpi.mpi_wtick()

#print the module version

print "      MYMPI VERSION",mpi.VERSION,"\n"
#test attributes
print "MPI_WTIME_IS_GLOBAL",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_WTIME_IS_GLOBAL)
print "  MPI_UNIVERSE_SIZE",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_UNIVERSE_SIZE)
print "      MPI_TAG_UB",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_TAG_UB)
print "      MPI_HOST",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_HOST)
print "      MPI_IO",mpi.mpi_attr_get(mpi.MPI_COMM_WORLD,mpi.MPI_IO)

print "%s%.1d%s%.1d" % ("mpi version ",mpi.MPI_VERSION,".",mpi.MPI_SUBVERSION)
parent=mpi.mpi_comm_get_parent()
if (parent == mpi.MPI_COMM_NULL):
print mpi.mpi_get_processor_name(),"running head "

copies=3

##### start up remote tasks #####
toRun=getcwd()+"/worker.py"
print mpi.mpi_get_processor_name(),"starting",toRun
newcom1=mpi.mpi_comm_spawn(toRun,"from_P_",copies,mpi.MPI_INFO_NULL,0,mpi.MPI_COMM_WORLD)
errors=mpi.mpi_array_of_errcodes()
print "errors=",errors
newcom1Size=mpi.mpi_comm_size(newcom1)
print "newcom1Size",newcom1Size," yes it is strange but it should be 1"
```

```

##### bcast #####
x=array([1,2,3,4]),"i")
count=4
print "head starting bcast",x
junk=mpi.mpi_bcast(x,count,mpi.MPI_INT,mpi.MPI_ROOT,newcom1)
print "head did bcast"

##### scatter #####
scat=array([10,20,30],"i")
junk=mpi.mpi_scatter(scat,1,mpi.MPI_INT,1,mpi.MPI_INT,mpi.MPI_ROOT,newcom1)

##### send/recv #####
for i in range(0,copies):
k=(i+1)*100
mpi.mpi_send(k,1,mpi.MPI_INT,i,1234,newcom1)
back=mpi.mpi_recv(1,mpi.MPI_INT,i,5678,newcom1)
print "from ",i,back

##### reduce #####
dummy=1000
final=mpi.mpi_reduce(dummy,1,mpi.MPI_INT,mpi.MPI_SUM,mpi.MPI_ROOT,newcom1)

sleep(5)

print "the final answer is=",final

toRun=getcwd()+"/worker"
print mpi.mpi_get_processor_name(),"starting",toRun
newcom2=mpi.mpi_comm_spawn(toRun,"from_C_",copies,mpi.MPI_INFO_NULL,0,mpi.MPI_COMM_WORLD)
errors=mpi.mpi_array_of_errcodes()
print "errors=",errors
newcom2Size=mpi.mpi_comm_size(newcom2)
print "newcom2Size",newcom2Size
sleep(15)

t2=mpi.mpi_wtime()
print "run time=",t2-t1,"with resolution=",tick

mpi.mpi_comm_free(newcom1)
mpi.mpi_comm_free(newcom2)
mpi.mpi_finalize()

```

**Example 7-2. worker.py**

```

#!/usr/bin/env python
import numpy
from numpy import *
import mpi
import sys
from time import gmtime, time,sleep

```

```

def stamp():
    timeTuple = gmtime(time())[1:6]
    return "%02d%02d%02d%02d%02d" % timeTuple

sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
myid=mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
numprocs=mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)
parent=mpi.mpi_comm_get_parent()
parentSize=mpi.mpi_comm_size(parent)
print "parentSize",parentSize

tod=stamp()
s=sys.argv[1]+"%2.2d" % myid
print "hello from python worker",myid," writing to ",s

x=array([5,3,4,2], 'i')
print "starting bcast"
buffer=mpi.mpi_bcast(x,4,mpi.MPI_INT,0,parent)
out = open(s, "w")
out.write(str(buffer))
out.write(tod+"\n")
out.close()

print myid," got ",buffer
junk=mpi.mpi_scatter(x,1,mpi.MPI_INT,1,mpi.MPI_INT,0,parent)
print myid," got scatter ",junk

back=mpi.mpi_recv(1,mpi.MPI_INT,0,1234,parent)
back[0]=back[0]+1
mpi.mpi_send(back,1,mpi.MPI_INT,0,5678,parent)

dummy=myid
final=mpi.mpi_reduce(dummy,1,mpi.MPI_INT,mpi.MPI_SUM,0,parent)

sleep(10)
mpi.mpi_comm_free(parent)
mpi.mpi_finalize()

```

**Example 7-3. worker.c**

```

#include <stdio.h>
#include <mpi.h>
#include <math.h>

void lam_darwin_malloc_linker_hack(){};
void timestmp(char *timestr);

int main(argc,argv)
int argc;

```

```

char *argv[];
{
    int myid, numprocs;
    FILE *f1;
    int i;
    int ierr;
    MPI_Comm parent;
    char fname[10];
    char stamp[11];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_get_parent(&parent);

    sprintf(fname,"%s%2.2d",argv[1],myid);

    printf("Hello from c worker %d writing to %s\n",myid,fname);
    f1=fopen(fname,"w");
    timestmp(stamp);
    fprintf(f1,"%s\n",stamp);
    fclose(f1);
    sleep(10);
    MPI_Comm_free(&parent);
    MPI_Finalize();
}

```

**Example 7-4. driver program for tspawn**

```

#include <time.h>
#include <sys/time.h>
#include <string.h>

void timestmp(char *timestr) {
/*      char timestr[11]; */
    struct timeval tp;
    struct timezone tzp;
    struct tm *tmstruct;
    time_t tod;
    gettimeofday(&tp,&tzp);
    tod=tp.tv_sec;
    tmstruct=gmtime(&tod);
/*      tmstruct=localtime(&tod); */
    sprintf(timestr,"%2.2d%2.2d%2.2d%2.2d%2.2d",tmstruct->tm_mon+1,
                                                    tmstruct->tm_mday,
                                                    tmstruct->tm_hour,
                                                    tmstruct->tm_min,
                                                    tmstruct->tm_sec);

    timestr[10]=(char)0;
}

```

## 7.8. Example Output

```
[peloton:~/mpi2] tkaiser% mpiexec -n 1 tspawn.py
MYMPI VERSION 1.13.0

MPI_WTIME_IS_GLOBAL 0
  MPI_UNIVERSE_SIZE 2
    MPI_TAG_UB 2147483647
      MPI_HOST 0
        MPI_IO 0
mpi version 2.0
peloton.sdsc.edu running head
peloton.sdsc.edu starting /Users/tkaiser/mpi2/worker.py
parentSize 3
hello from python worker 2  writing to  from_P_02
starting bcast
parentSize 3
hello from python worker 1  writing to  from_P_01
starting bcast
errors= [0 0 0]
newcom1Size 1  yes it is strange but it should be 1
head starting bcast [1 2 3 4]
parentSize 3
hello from python worker 0  writing to  from_P_00
starting bcast
head did bcast
2  got  [1 2 3 4]
1  got  [1 2 3 4]
0  got  [1 2 3 4]
0  got scatter  [10]
from 0 [101]
2  got scatter  [30]
1  got scatter  [20]
from 1 [201]
from 2 [301]
the final answer is= [3]
peloton.sdsc.edu starting /Users/tkaiser/mpi2/worker
errors= [0 0 0]
newcom2Size 1
Hello from c worker 1 writing to from_C_01
Hello from c worker 2 writing to from_C_02
Hello from c worker 0 writing to from_C_00
run time= 21.964277029 with resolution= 1e-06
[peloton:~/mpi2] tkaiser%
```

## 7.9. A simple MPI-2 example using two independent programs

The sources `main0.py` and `main1.py` show how to use the MPI-2 commands:

- **mpi\_open\_port**
- **mpi\_close\_port**
- **mpi\_comm\_accept**
- **mpi\_comm\_connect**
- **mpi\_comm\_disconnect**

These commands allow two separate MPI jobs to connect to each other. These examples are designed to be run interactively. Start one copy of each application in its own window. The `>main1.py` program will write a `port_name` string to the terminal, something like:

```
port= tag#0$port#54838$description#192.31.21.33$ifname#192.31.21.33$
```

The other program, `main0.py`, takes this string as input. It will prompt using:

```
port_name>
```

Copy the string from the first window to the second inclosing it in quotes. After the two programs connect they send messages to each other and quit.

#### Example 7-5. `main0.py`

```
#!/usr/bin/env python
import numpy
from numpy import *
import mpi
import sys
from time import sleep

sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
myid=mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
numprocs=mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)

print "hello from python main0   myid= ",myid

port_name=input("port_name>")
print "port=",port_name
server=mpi.mpi_comm_connect(port_name,mpi.MPI_INFO_NULL,0,mpi.MPI_COMM_WORLD)

back=array([100],"i")
mpi.mpi_send(back,1,mpi.MPI_INT,0,5678,server)
back=mpi.mpi_recv(1,mpi.MPI_INT,0,1234,server)
print "got back=",back

sleep(10)
mpi.mpi_comm_disconnect(server);
mpi.mpi_finalize()
```

**Example 7-6. main1.py**

```
#!/usr/bin/env python
import numpy
from numpy import *
import mpi
import sys
from time import sleep

sys.argv = mpi.mpi_init(len(sys.argv),sys.argv)
myid=mpi.mpi_comm_rank(mpi.MPI_COMM_WORLD)
numprocs=mpi.mpi_comm_size(mpi.MPI_COMM_WORLD)

print "hello from python main1  myid= ",myid

port_name=mpi.mpi_open_port(mpi.MPI_INFO_NULL);
print "port=",port_name
client=mpi.mpi_comm_accept(port_name,mpi.MPI_INFO_NULL,0,mpi.MPI_COMM_WORLD)

back=mpi.mpi_recv(1,mpi.MPI_INT,0,5678,client)
print "back=",back
back[0]=back[0]+1
mpi.mpi_send(back,1,mpi.MPI_INT,0,1234,client)

sleep(10)
mpi.mpi_close_port(port_name);
mpi.mpi_comm_disconnect(client);
mpi.mpi_finalize()
```

# Chapter 8. Version 1.11 Release Notes

## 8.1. Introduction

November 18, 2005

MYMPI version 1.11 is a maintenance release. It also added a function to return the copywrite, `mpi.copywrite()`.

There are a few differences between MYMPI version 1.0 and version 1.1. The main differences are related to:

1. Long command line argument lists are allowed
2. `mpi_init` now returns the command line arguments
3. Contains a version number
4. Contains a workflow example

## 8.2. Long command line argument lists are allowed

LAM MPI does not like long command line argument lists. MYMPI was crashing when command line arguments lists were a little more than 100 characters long. The module was changed to work around this issue. Users of other MPI packages should not see any differences.

## 8.3. `mpi_init` now returns the command line arguments

The command `mpirun` can add command line arguments that are passed to your program. The MPICH (P4) version of `mpirun` does this. These extra command line arguments are used to help start the parallel job. When the command line is passed to `MPI_Init` the extra arguments are removed.

The point is that an application might not get the expected command line arguments. Accessing the command line arguments before and after `MPI_init` is called might not return the same thing.

As of version 1.1 the routine `mpi.mpi_int` will return the command line arguments after `MPI_Init` is called. Thus the preferred usage of this call is now

```
import sys
sys.argv = mpi_init(len(sys.argv), sys.argv)
```

After these calls `sys.argv` will contain the expected command line arguments

## 8.4. Contains a version number

The constant `mpi.VERSION` is now defined as '1.1'.

## 8.5. Contains a work flow example

The directory `workit` contains a simple workflow example.

`workit.py` is a simple pipeline processing or "workflow" program. `workit.py` is used when you need to perform a series of sequential operations on data sets. It uses the `mypmi` Python MPI module to spread work across multiple processors. MPI is used for synchronization and convenience of launching multiple processors. This example also shows how you can pass character data using the module.

See the notes in the directory for additional information.